Clase 4 - Recursión

---

1. Introducción a la Recursión

¿Qué es la recursión? La recursión es una forma de definir una función o un algoritmo en términos de sí mismo. Esto significa que la función o el algoritmo se llama a sí mismo con diferentes argumentos o datos hasta llegar a un caso base, que es el caso más simple o trivial que se puede resolver directamente sin necesidad de llamarse a sí mismo. La idea es que al resolver el caso base y las llamadas anteriores, se pueda obtener la solución del problema original.

La **recursión** es una técnica de programación donde una función se llama a sí misma para resolver subproblemas más pequeños de un problema más grande. Este enfoque es especialmente útil para problemas que pueden dividirse en subproblemas más pequeños y similares.

Según **Luis Joyanes Aguilar**, en su libro *Fundamentos de Programación*, la recursión se puede entender como un proceso que "permite definir soluciones de problemas a través de la división del problema en subproblemas de la misma naturaleza".

En términos generales, un algoritmo recursivo consta de dos partes:

- **Caso base**: La condición de parada del algoritmo, donde se define el resultado sin realizar una llamada recursiva.
- **Caso recursivo**: La parte del algoritmo donde se llama a la función a sí misma con argumentos modificados para alcanzar eventualmente el caso base.

La importancia de los casos general y base radica en que permiten definir la solución de un problema complejo en términos de sí mismo, pero reduciendo su tamaño o dificultad en cada paso. Además, los casos base aseguran que la recursión termina en algún momento, evitando así un bucle infinito o un desbordamiento de memoria.

Para encontrar los casos general y base de un problema recursivo, se puede seguir el siguiente método:

- Identificar el problema que se quiere resolver y los datos o argumentos que necesita}.
- Pensar en cómo se puede dividir el problema en subproblemas más pequeños o simples de la misma naturaleza.
- Determinar si hay algún subproblema que se pueda resolver directamente, sin necesidad de usar la función recursiva. Ese será el caso base.
- Determinar cómo se puede resolver el problema original a partir del resultado del subproblema más pequeño o simple. Ese será el caso general.

Escribir la función recursiva usando una estructura condicional que distinga entre el caso base y el caso general.

Para ilustrar estos conceptos con ejemplos, vamos a usar dos funciones recursivas clásicas: la función factorial y fibonacci .

La función factorial calcula el producto de todos los números enteros positivos desde 1 hasta un número n dado. La función factorial se puede expresar matemáticamente como:

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

En este ejemplo, el caso base es cuando n es cero, y se devuelve 1. El caso general es cuando n es mayor que cero, y se devuelve n por el factorial de n-1. Así, podemos escribir la función factorial en cualquier lenguaje de programación usando una estructura condicional.

```python
def factorial(n):
    if n == 0:
        return 1  # Caso base
    else:
        return n * factorial(n - 1)  # Caso recursivo

# Ejemplo de uso
print(factorial(5))  # Salida: 120
```

Para entender cómo funciona la recursión, podemos seguir el flujo de ejecución de la función con un ejemplo concreto. Supongamos que queremos calcular el factorial de 4. Entonces, llamamos a la función con el argumento 4:

factorial(4)

Como 4 no es cero, entramos en el caso recursivo y devolvemos 4 por el resultado de llamar a la función con el argumento 3:

*factorial(4) = 4 * factorial(3)*

Ahora tenemos que calcular el valor de factorial(3). Como 3 tampoco es cero, entramos otra vez en el caso recursivo y devolvemos 3 por el resultado de llamar a la función con el argumento 2:

*factorial(3) = 3 * factorial(2)*

Seguimos con el mismo proceso hasta llegar al caso base, donde n es cero y devolvemos 1:

*factorial(2) = 2 * factorial(1)*
*factorial(1) = 1 * factorial(0)*
factorial(0) = 1

Ahora podemos sustituir los valores desde abajo hacia arriba y obtener el resultado final:
*factorial(4) = 4 * factorial(3)*
*factorial(4) = 4 * (3 * factorial(2))*
*factorial(4) = 4 * (3 * (2 * factorial(1)))*
*factorial(4) = 4 * (3 * (2 * (1 * factorial(0))))*
*factorial(4) = 4 * (3 * (2 * (1 * 1)))*
*factorial(4) = 4 * (3 * (2 * 1))*

*factorial(4) = 4 * (3 * 2)*
*factorial(4) = 4 * 6*
factorial(4) = 24

Como podemos ver, la recursión nos permite resolver un problema complejo con una solución simple y elegante. Sin embargo, la recursión también tiene sus desventajas, como veremos más adelante.

Otro ejemplo de una función recursiva es la función de Fibonacci, que calcula el n-ésimo término de la serie de Fibonacci. La serie de Fibonacci es una sucesión de números naturales que empieza con 0 y 1, y donde cada término es la suma de los dos anteriores. Los primeros términos de la serie son:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

La definición matemática del n-ésimo término de la serie es la siguiente:

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{si } n > 1 \end{cases}$$

En este ejemplo, los casos base son cuando n es cero o uno, y se devuelve el valor de n. El caso general es cuando n es mayor que uno, y se devuelve la suma del resultado de llamar a la función con los argumentos n-1 y n-2. Así, podemos escribir la función de Fibonacci en cualquier lenguaje de programación usando una estructura condicional

Ejemplo en Python: Fibonacci

El cálculo de la secuencia de Fibonacci se puede hacer de manera recursiva. Sin embargo, esta implementación no es la más eficiente:

```python
def fibonacci(n):
    if n <= 1:
        return n  # Casos base
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Casos recursivos

# Ejemplo de uso
print(fibonacci(5))  # Salida: 5
```

2. Recursión Directa e Indirecta

La recursión se puede clasificar en dos tipos según cómo se realice la llamada recursiva: recursión directa e indirecta. La recursión directa ocurre cuando una función se llama a sí misma dentro de su cuerpo. Es el tipo más común y simple de recursión. Los ejemplos anteriores de las funciones factorial y de Fibonacci son ejemplos de recursión directa.
La recursión indirecta ocurre cuando una función se llama a sí misma a través de otra función. Es decir, hay una cadena de llamadas entre funciones que termina volviendo a la función original.

La recursión entonces puede clasificarse en dos tipos principales:

- **Recursión Directa**: Una función se llama a sí misma directamente. Este es el tipo más común de recursión. Un ejemplo típico es el cálculo del factorial que hemos visto anteriormente.
- **Recursión Indirecta**: Una función A llama a otra función B, la cual eventualmente llama de nuevo a la función A. Este tipo de recursión es menos común y puede ser más difícil de seguir y depurar.

Introducción a Algoritmos por **Thomas H. Cormen y otros** explica que "la recursión indirecta ocurre cuando un conjunto de dos o más funciones se llaman entre sí en un ciclo".

Ejemplo en Python: Recursión Indirecta

```python
def function_A(n):
    if n > 0:
        print(n, " en function_A")
        function_B(n - 1)

def function_B(n):
    if n > 0:
        print(n, " en function_B")
        function_A(n - 1)

# Ejemplo de uso
function_A(3)
```

Salia del ejemplo:
3 en function_A
2 en function_B
1 en function_A

3. Ventajas y Desventajas de la Recursión

Ventajas

- **Simplicidad**: Las soluciones recursivas son a menudo más sencillas y más fáciles de entender y escribir. En problemas complejos, como los que involucran estructuras de datos dinámicas (árboles, grafos), la recursión puede ofrecer una solución más intuitiva y directa.
- **Menos código**: Las funciones recursivas pueden reducir significativamente la cantidad de código necesario en comparación con las soluciones iterativas.

Según **Gustavo López, Ismael Jeder, y Augusto Vega** en *Análisis y Diseño de Algoritmos*, "la recursión simplifica la programación de algoritmos que de otro modo requerirían una manipulación compleja de estructuras de datos y estados intermedios".

Desventajas

- **Consumo de memoria**: Cada llamada recursiva se almacena en la pila de llamadas. Para problemas con un alto número de llamadas recursivas, esto puede llevar a un desbordamiento de pila (stack overflow).

- **Eficiencia**: Las soluciones recursivas pueden ser menos eficientes que sus equivalentes iterativas debido a la sobrecarga de las llamadas a funciones y el consumo adicional de memoria.

Como se menciona en **Introduction to Algorithms**, "el uso excesivo de la recursión puede llevar a ineficiencias significativas, especialmente si no se maneja adecuadamente el caso base y la memoria".